Escuela Politécnica Superior

Diseño de un agente inteligente para videojuegos tipo Arcade

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

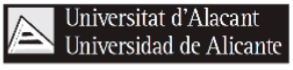
Autor:
Benjamín Pamies Cartagena

Tutor/es:
Patricia Compañ Rosique
Rosana Satorre Cuerda

Septiembre 2019

Universitat d'Alacant
Universidad de Alicante

Diseño de un agente inteligente para videojuegos tipo Arcade



TRABAJO FIN DE MASTER
Máster en Ingeniería Informática

Autor:
Benjamín Pamies Cartagena

Coordinadoras:
Patricia Compañ Rosique
Rosana Satorre Cuerda

Departamento de Ciencia de la Computación e Inteligencia Artificial
Escuela Politécnica Superior
Universidad de Alicante

Septiembre 2019

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

Resumen

La inteligencia artificial es una rama de las ciencias de la computación que estudia algoritmos que imitan capacidades inteligentes con el objetivo de resolver tareas complejas que no son posibles para algoritmos tradicionales. Los videojuegos, además de servir como actividad de ocio para el entretenimiento en general, proporcionan un contexto ideal para probar algoritmos inteligentes. En este trabajo se ha diseñado un agente inteligente capaz de jugar a juegos de tipo Arcade, concretamente de la consola Atari 2600. Para ello, se han analizado distintas técnicas utilizadas frecuentemente en aplicaciones de inteligencia artificial que resuelven problemas como el de los videojuegos.

El agente desarrollado realiza interacciones con el entorno al que se enfrenta en cada juego y aprende a realizar los movimientos que le garantizan aumentar sus probabilidades de victoria a largo plazo. Este aprendizaje se basa en maximizar las recompensas que el agente recibe durante su entrenamiento, en lo que se conoce como aprendizaje reforzado.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Contexto

Este trabajo se enmarca dentro del estudio de la Inteligencia Artificial, que es aquella inteligencia adoptada por una máquina. Cuando hablamos de inteligencia en máquinas nos referimos a la capacidad de un computador de realizar tareas que requieren análisis, lógica, percepción del entorno, razonamiento, aprendizaje y otras capacidades que se consideran parte de lo que se define como inteligente.

En la actualidad, la inteligencia en computación se compone de técnicas que imitan muchas de estas capacidades a fin de resolver tareas muy concretas. Es lo que se conoce como inteligencia de máquina débil: pequeños agentes inteligentes que realizan labores propias de los seres humanos simulando inteligencia. Existen ejemplos de este tipo de algoritmos inteligentes como, por ejemplo, robots que realizan tareas logísticas o, en actividades tan importantes como la economía o la mediciona, agentes inteligentes que toman decisiones fundamentales basándose en experiencias pasadas. Hipotéticamente, se conoce también como inteligencia de máquina fuerte, igualmente conocida como Inteligencia Artificial General, a aquella inteligencia artificial que puede resolver cualquier tarea intelectual incluso superando la inteligencia humana. Existe un gran debate en la comunidad científica sobre si las máquinas pueden llegar algún día a ser realmente inteligentes y las consecuencias éticas y morales que supondría.

Al márgen de debates morales, este trabajo se centra en el Aprendizaje Automático, un campo de la inteligencia artificial que pretende imitar la capacidad de aprender en las máquinas. La investigación en aprendizaje automático se centra en la elaboración de algoritmos que mejoran su desempeño en una tarea a partir de la experiencia. Las experiencias pueden ser datos aportados al algoritmo previamente o incluso pueden ser observaciones del entorno en el que el algoritmo es ejecutado. De cualquier forma, se considera que un agente aprende cuando adquiere conocimiento por sí mismo.

Concretamente, en este trabajo se propone el reto de crear un agente inteligente que sea capaz de aprender a jugar a juegos Atari. Los videojuegos son un entorno de ejecución óptimo para probar y estudiar la capacidad de aprender de un algoritmo inteligente.

1.2. Justificación

La implementación de un agente inteligente que sea capaz de jugar eficientemente a juegos de tipo Arcade requiere el estudio y comprensión de técnicas dentro del campo del aprendizaje automático, también conocido por su nombre en inglés *machine learning*. Dentro del machine learning es habitual utilizar los videojuegos como un problema de estudio para el desarrollo de nuevos métodos y técnicas de construcción de inteligencia artificial, sobre todo técnicas de aprendizaje por refuerzo, una de las áreas del aprendizaje automático que es ampliamente explicada a lo largo de esta memoria.

La motivación principal de este trabajo es la de adquirir conocimientos en aprendizaje automático, y en especial en apredizaje por refuerzo, aplicados en el interesante mundo de los videojuegos pero que puedan ser extensibles a otros problemas diferentes y servir como base para una especialización en este campo.

1.3. Estructura de capítulos

Esta memoria está dividida en los capítulos descritos a continuación:

- El capítulo 2 hace mención a los elementos conceptuales que sirven de base para la realización del trabajo. En este capítulo se redactan los estudios previos que se han realizado, relacionados con el problema planteado.
- En el capítulo 3 se enumeran los objetivos generales que se quieren alcanzar con este trabajo.
- El capítulo 4 pasa a describir cómo se ha llevado a cabo la realización del presente trabajo. Se han descrito las técnicas y procedimientos que se han utilizado, esto es, las fases que se han seguido durante el desarrollo del trabajo, así como los intrumentos o herramientas de las que se han hecho uso en cada una de las distintas fases.
- El capítulo 5 aborda el contenido práctico del trabajo. Se detalla el desarrollo e implementación del agente inteligente, explicando los algoritmos implementados. Se explica la evolución que ha seguido el agente durante su desarrollo, las pruebas que se han realizado, el proceso de entrenamiento que ha seguido el agente final y los experimentos finales

realizados sobre el agente. En este capítulo se incluyen también los resultados que se han obtenido en los experimentos, así como el análisis y la discusión de los mismos.

- Por último, en el capítulo 6 se exponen las conclusiones a las que se han llegado realizando este trabajo.

Capítulo 2

Estado del arte

En este capítulo se realiza un estudio de cómo se encuentra actualmente el estado del arte del aprendizaje automático en los videojuegos. En primer lugar, se hace un breve repaso por la historia de la inteligencia artificial en los videojuegos, desde sus comienzos hasta la actualidad. Seguidamente se explican diferentes entornos o *frameworks* para la investigación en este campo y, por último, se explican las bases teóricas del aprendizaje por refuerzo, ténica utilizada para el desarrollo de agentes inteligentes en ambientes como el de los videojuegos.

2.1. Evolución de la IA para la resolución de juegos

A lo largo de la historia de la inteligencia artificial (IA) se han utilizado juegos de diferentes tipos y dificultad como entornos de prueba donde desarrollar y evolucionar algoritmos en este campo. Para resolver un juego es necesario utilizar la imaginación, crear estrategias y usar una serie de habilidades asociadas a la inteligencia humana, viéndose estos juegos como excelentes *benchmark* para evaluar el rendimiento de las técnicas utilizadas para resolver ciertos problemas. Estas técnicas y algoritmos desarrollados y verificados en entornos de simulación de juegos son luego utilizados en el mundo real para ser aplicados en problemas más importantes como la medicina o la robótica.

Es por eso que los grandes laboratorios de inteligencia artificial como OpenAI y DeepMind han creado sus propias plataformas para que el resto de la comunidad científica experimente nuevos algoritmos de aprendizaje automático sobre una gran variedad de juegos.

2.1.1. Juegos de mesa

Los primeros algoritmos que se enfrentaron a juegos fueron programas diseñados para jugar a las damas, siendo el más destacado Chinook, desarro-

llado en la Universidad de Alberta, que logró clasificar para el Campeonato Mundial frente a humanos.

Pero el primer gran hito de la IA en la resolución de juegos se produjo en 1997 cuando Deep Blue, desarrollado por IBM, venció al campeón del mundo de ajedrez Garri Kaspárov. Este evento fue muy mediatizado y supuso una proliferación de los denominados motores de ajedrez, programas capaces de jugar al ajedrez a un nivel muy superior a los mejores jugadores del mundo. El motor de ajedrez Stockfish fue el primero en superar los 3500 puntos Elo[1], muy superior al actual récord humano de 2882 puntos ostentado por el también campeón del mundo Magnus Carlsen.

Estos motores de ajedrez, al igual que los programas que juegan a las damas, utilizan información complementaria como base de datos de situaciones típicas del juego y libros de aperturas. Suelen basarse en estrategias de búsqueda en profundidad y algoritmos minimax.

En 2015, la empresa DeepMind hizo público un nuevo sistema inteligente, llamado AlphaGo, capaz de jugar al juego de mesa Go. El Go es un juego que tiene un número inabarcable de posibles situaciones tras cada movimiento, por lo que los desarrolladores combinaron un árbol de búsqueda con un sistema de aprendizaje profundo con redes neuronales. De una forma parecida a lo ocurrido con Deep Blue y Kaspárov, el sistema AlphaGo fue enfrentado a cinco partidas contra el considerado mejor jugador de Go de la historia, el surcoreano Lee Se-dol. El enfrentamiento fue mundialmente televisado y generó gran espectación. Finalmente AlphaGo venció por 4-1.

Poco después DeepMind lanzó una versión mejorada de AlphaGo, llamada AlphaZero. Esta nueva versión no aprende a partir de partidas existentes como hace AlphaGo, sino que lo hace enfrentándose contra sí mismo, sin acceso a libros de apertura o base de datos de tablas finales. AlphaZero se creó con un enfoque generalizado, capaz de jugar a Go, ajedrez y shogi. AlphaZero fue enfrentado a Stockfish en 100 partidas de ajedrez, con el resultado de 28 victorias, 72 tablas y ninguna derrota.

2.1.2. Juegos Atari

A medida que la IA iba resolviendo juegos, los investigadores iban buscando nuevo retos. Cada vez que un agente inteligente aprendía a jugar a juegos que parecían imposibles de resolver para una máquina, se afrontaba un juego todavía más difícil. En la última década, la investigación en IA se ha adentrado en el mundo de los videojuegos.

A diferencia de los juegos de mesa, donde el entorno a explorar por el agente está limitado a un tablero, en los videojuegos el entorno es muchísimo más extenso, complejo y con una grandísima variedad de posibles elemen-

[1]El sistema de puntuación Elo es el método utilizado en ajedrez y otros deportes mentales para medir la habilidad de los jugadores.

tos: obstáculos, vidas, enemigos, llaves y cerraduras, etc. Esto hace de los videojuegos un reto aún mayor.

Los primeros videojuegos a los que se enfrentaron los investigadores fueron juegos de tipo *arcade*, concretamente juegos Atari. La empresa OpenAI creó una plataforma para el desarrollo de inteligencia general, en la que poder evaluar los algoritmos sobre entornos emulados de videojuegos Atari, y la publicó como código abierto para el uso de la comunidad científica. Tras esto, aparecieron muchas publicaciones científicas sobre nuevas técnicas de inteligencia artificial que demostraban sus resultados sobre esta plataforma, consiguiendo puntuaciones superiores a las de los humanos.

2.1.3. Actualidad

Una vez los juegos Atari se consideran en su mayoría resueltos para la IA, el nuevo paso son videojuegos más complejos. Recientemente DeepMind ha creado AlphaStar, una IA capaz de jugar al videojuego StarCraft II. Se trata de un videojuego de estrategia en tiempo real donde el objetivo es derrotar al oponente sobre un mapa en el que hay que realizar tareas como construir edificios en la base, recolectar recursos o crear unidades de combate.

Existen diferencias entre este tipo de juego y los juegos de mesa y Atari, que lo hacen mucho más complejo de resolver. La diferencia con los juegos de mesa es que no se trata de un juego por turnos, sino que las acciones de ambos jugadores ocurren de manera simultánea. Además, en un juego de mesa en todo momento puede verse la posición completa del tablero, por lo que lo convierte en un juego de información completa. En juegos de información incompleta como StarCraft II los jugadores en ocasiones tienen que idear estrategias sin conocer qué está realizando el oponente. La diferencia con los juegos Atari es que el agente debe enfrentarse a humanos y no contra la máquina, lo que lo hace mucho más imprevisible.

A finales de 2018, AlphaStar se enfretó a dos jugadores profesionales de StarCraft II, ganando todas las partidas (en la figura 2.1 puede verse una captura de pantalla de una de las partidas).

En abril de 2019, OpenAI Five se convirtió en la primera IA en vencer a los campeones del mundo en un juego de deportes electrónicos, o en inglés *e-sports*, tras derrotar a los campones del mundo del también juego de estrategia Dota 2.

2.2. Plataformas de desarrollo

En el mundo de la investigación de la IA en los videojuegos se utilizan plataformas de simulación de juegos en las que poder probar los agentes inteligentes. Estas plataformas permiten a los investigadores centrarse en el diseño e implementación del agente sin necesidad de tener que implementar

Figura 2.1: Fotograma de una partida de AlphaStar contra LiquidTLO.

también el entorno de pruebas en el que ejecutarlo.

Existen muchas de estas plataformas, entras ellas, dos que son descritas a continuación: *The Arcade Learning Environment* y *OpenAI Gym*.

2.2.1. The Arcade Learning Environment

The Arcade Learning Environment (ALE) es una plataforma de evaluación de agentes inteligentes sobre juegos de la consola Atari 2600. Entre los juegos disponibles en ALE se encuentran algunos como *Montezuma's Revenge*, *Enduro*, *Space Invaders* y *Ms. Pacman*.

Los desarrolladores de la plataforma ALE la plantean tanto como un problema de desafío como una plataforma de evaluación (Bellemare et al., 2013). Fue creada con el objetivo de permitir el diseño de agentes inteligentes que resuelvan tantos juegos como sea posible, sin requerir información específica de cada uno de ellos. La plataforma admite una variedad de configuraciones de problemas diferentes y ha estado recibiendo una atención importante por parte de la comunidad científica.

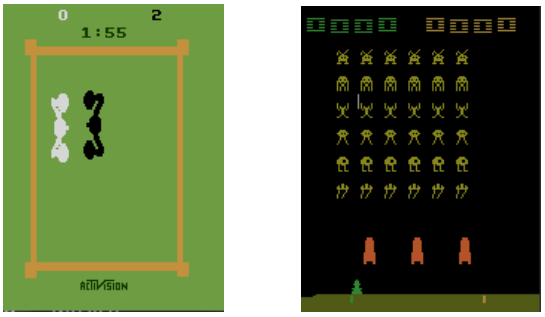
ALE está desarrollada sobre Stella, un emulador Atari 2600. Lo que hace el framework por encima es transformar cada juego en un problema de aprendizaje por refuerzo al identificar las puntuaciones y si el juego ha finalizado. En ALE es necesario descargar la ROM de un juego para emularlo en la plataforma y así poder probar el agente sobre él.

Según Machado et al. (2018), los juegos de Atari 2600 son entornos excelentes para evaluar agentes por tres razones: 1) son lo suficientemente variados para proporcionar múltiples tareas diferentes, que requieren competencia general, 2) son interesantes y desafiantes para los humanos, y 3) no tienen sesgo de experimentador, al haber sido desarrollado por un grupo independiente.

2.2.2. OpenAI Gym

Gym es un conjunto de herramientas para desarrollar y comparar algoritmos de aprendizaje por refuerzo, desarrolladas por la empresa de inteligencia artificial OpenAI. También permite el diseño de soluciones generales.

En Gym existen varios tipos de entornos diferentes, desde más complejos a más sencillos, que involucran muchos tipos diferentes de datos. Están los problemas clásicos del aprendizaje por refuerzo como *CartPole* o *MountainCar*, simulación de robots 2D y 3D utilizando el motor de física *MuJoCo*[2] y, por supuesto, juegos Atari como los que pueden verse en la figura 2.2. Los juegos Atari en Gym son simulados a través de ALE, con la diferencia de que en Gym los juegos vienen integrados en el propio framework.



(a) Boxing (b) Spce Invaders

Figura 2.2: Ejemplos de juegos Atari en Gym.

Cuando el agente realiza un movimiento en alguno de los entornos de Gym, la plataforma le devuelve al agente los siguientes valores:

- Un objeto llamado *observation* que representa la obervación del entorno. En un juego de mesa, por ejemplo, la observación sería el estado del tablero.
- Un número llamado *reward* que representa la recompensa que recibe el agente mientras realiza movimientos en el entorno.

[2] A pesar de que Gym es de código abierto, MuJoCo es software propietario, se requiere de una licencia de MuJoCo para simular robots en Gym.

- Un booleano llamado *done* que especifica si el agente tiene que reiniciarse o si ha completado con éxito el juego. Puede ser verdadero, por ejemplo, cuando te quedas sin vidas.
- Un diccionario llamado *info* que devuelve información relevante sobre el entorno. Esta información no puede ser utilizada por el agente sino que es información para el desarrollador.

Estos valores son utilizados por el agente para mejorar su desempeño en los siguientes pasos que ejecute. Principalmente, el objetivo es mejorar las recompensas recibidas basándose en las observaciones. Esto es exactamente en lo que consiste el aprendizaje por refuerzo, que viene explicado en el siguiente apartado.

2.3. Aprendizaje por refuerzo

Existen varios tipos de aprendizaje automático. Entre ellos se encuentran el aprendizaje supervisado y no supervisado, dos técnicas muy utilizadas para el análisis de datos. En estas técnicas el sistema necesita ser alimentado con una gran cantidad de datos, proporcionados por humanos, a partir de los cuales el sistema aprende a extraer características útiles e información relevante. Esta información, que un humano sería incapaz de deducir de una cantidad tan ingente de datos, puede ser utilizada para, por ejemplo, predicciones. Pero para entornos como los videojuegos el sistema necesita obtener los datos del propio entorno mientras interactúa con él. En este caso la estrategia más utilizada es que el sistema aprenda a interactuar con el entorno mediante refuerzos.

El aprendizaje reforzado o por refuerzo (del inglés *reinforcement learning*), a partir de ahora RL, es un tipo de aprendizaje automático mediante el cual un agente inteligente aprende en base a recompensas o refuerzos, que pueden ser positivos o negativos dependiendo de si la acción realizada por el agente le acerca a alcanzar su meta u objetivo.

En RL, el agente interactúa (realiza acciones) sobre el entorno o medio ambiente y de esta interacción recibe información que le ayuda a desenvolverse mejor en el ambiente con el paso del tiempo. En cada interacción, el agente se encuentra en un estado s, de un conjunto de todos los posibles estados S, $s \epsilon S$, y realiza una acción a, de un conjunto de todas las posibles acciones A, $a \epsilon A$. Tras realizar la acción el agente pasa a estar en un nuevo estado s' y recibe del ambiente una recompensa r. Este proceso puede verse representado en la figura 2.3.

El agente debe realizar aquellas acciones que aumenten la suma total de recompensas que recibe, es decir, tiene que encontrar una política de movimientos que maximice a largo plazo el refuerzo acumulado. Una política π es un mapeo de estados a acciones que determina la probabilidad $\pi(a|s)$ de

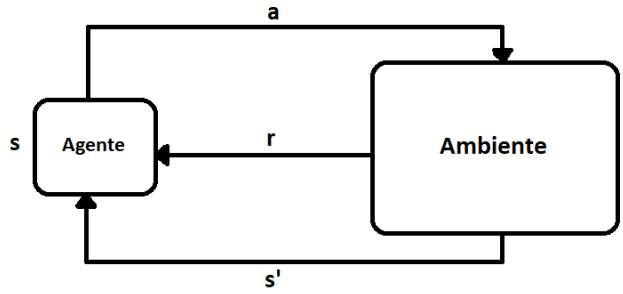


Figura 2.3: Esquema de entrenamiento reforzado.

realizar una acción a encontrándose en un estado s. Este mapa es actualizado en base a la experiencia que adquiere el agente durante el entrenamiento.

En general, el medio ambiente es no determinista, es decir, tomar la misma acción en el mismo estado puede dar resultados diferentes. Otra característica importante es que la recompensa recibida para cada par estado-acción es independiente de las acciones realizadas con anterioridad. Esto satisface la propiedad de Markov que dice que el futuro de un estado solo se relaciona con el pasado a través del estado "presente". Una vez que se decide el estado actual, el futuro no tiene relación con los estados pasados (Zhang et al., 2010).

2.3.1. Proceso de decisión de Markov

Un proceso de decisión de Markov (del inglés *Markov Decision Process*), a partir de ahora MDP, es un procedimiento que se da en entornos no deterministas o con cierto grado de incertidumbre que utiliza una función de transición de estados $T(s'|s,a)$ encargada de calcular la probabilidad de alcanzar un nuevo estado al realizar una acción sobre el estado actual.

Formalmente, la función T determina la probabilidad de alcanzar el estado s' al realizar la acción a desde el estado s y se define como

$$T(s'|s,a) = Pr^{a}_{s,s'} = Pr(s_{t+1} = s'|s_t = s, a_t = a) \tag{2.1}$$

Además de la función de transición (T), MDP consta de una función de recompensa $R(s,a)$ encargada de calcular el valor de la recompensa r que espera el agente recibir al realizar la acción a estando en el estado s y se define como

$$R(s,a) = R^{a}_{s} = E(r_{t+1}|s_t = s, a_t = a) \tag{2.2}$$

La meta u objetivo del agente inteligente es, por tanto, aprender la política de acción π que maximice el refuerzo medio esperado. Para esto, se

dispone de unas funciones adicionales, las funciones de valor, con las que se puede medir qué tan bueno es el estado en el que se encuentra el agente y cómo de bueno es realizar una acción en un estado concreto. La función de valor de estado $V^{\pi}(s)$ determina el retorno esperado de seguir la política π desde el estado s. La función de valor de acción $Q^{\pi}(s,a)$ determina el retorno esperado de seleccionar la acción a en el estado s y entonces seguir la política π (Jeerige et al., 2019). Estas funciones se basan en las recompensas que se reciben siguiendo una política de movimientos. Es natural pensar que el total de recompensas R esperadas a partir de un estado es la suma de las recompensas que se esperan obtener en el futuro. Si en cada tiempo $t = 0, 1, 2, 3...$ se recibe una recompensa $r_t = r_0, r_1, r_2, r_3...$, la recompensa total acumulada estaría definida como el sumatorio de

$$R = r_0 + r_1 + r_2 + r_3 + ... = \sum_{t=0}^{\infty} r_t \tag{2.3}$$

El problema de hacer esta suma no ponderada de las recompensas es que no converge a ningún número y por lo tanto no converge a ninguna solución. Para resolver este problema, se aplica un factor de descuento $\gamma\epsilon(0,1]$ encargado de priorizar las recompensas más inmediatas frente a las recompensas más alejadas en el tiempo. Con este factor el sumatorio queda como

$$R = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + ... = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.4}$$

Esta suma ponderada exponencialmente se utiliza para determinar la recompensa total esperada estando en el estado s y siguiendo la política π, es decir, la función de valor de estado $V^{\pi}(s)$ mencionada anteriormente que al aplicar la suma descontada de los refuerzos queda como

$$V^{\pi}(s) = E_{\pi}(R_t|s_t = s) = E_{\pi}(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s) \tag{2.5}$$

De igual forma, el valor esperado tomando una acción a en el estado s bajo la política π, es decir, la función de valor de acción $Q^{\pi}(s,a)$ queda como

$$Q^{\pi}(s,a) = E_{\pi}(R_t|s_t = s, a_t = a) = E_{\pi}(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = t) \tag{2.6}$$

Para obtener la política π que maximice las recompensas a largo plazo, en otras palabras, obtener la política de acción óptima, a la que denotaremos como π^*, hay que utilizar las funciones de valor óptimas V^* y Q^* que se obtienen maximizando V y Q de manera que $V^*(S) = \text{máx}_{\pi} V^{\pi}(s)$ y $Q^*(s,a) = \text{máx}_{\pi} Q^{\pi}(s,a)$. Estas funciones de valor óptimas se pueden considerar subdivisiones de la ecuación de optimalidad de Bellman (1958).

Esta ecuación es la suma de la recompensa en el estado actual más la suma descontada de recompensas esperadas tras escoger la acción óptima.

La ecuación de Bellman es de la forma

$$V^*(s) = \max_{a \epsilon A(s)} \sum_{s'} Pr^a_{s,s'}[R(s,a) + \gamma V^*(s')] \tag{2.7}$$

Donde $V^*(s) = \max_{a \epsilon A(s)} Q^*(s,a)$, por lo que

$$Q^*(s,a) = \sum_{s'} Pr^a_{s,s'}[R(s,a) + \gamma V^*(s')] \tag{2.8}$$

O también

$$Q^*(s,a) = \sum_{s'} Pr^a_{s,s'}[R(s,a) + \gamma \max_{a' \epsilon A(s)} Q^*(s',a')] \tag{2.9}$$

2.3.2. Programación dinámica

Para aquellos problemas en los que se conocen todos los conceptos del apartado anterior: todos los estados posibles, acciones, las funciones T y R, etc., a saber, se tiene un conocimiento completo del MDP, se utiliza programación dinámica para resolverlos.

Volviendo a la ecuación 2.7 de Bellman, esta obtiene el valor máximo de la suma de la recompensa del estado actual más el valor máximo de las recompensas futuras a partir de ese estado, en todos los posibles estados que se pueden alcanzar desde el estado actual al aplicar una acción. A su vez, el máximo de las recompensas futuras se calcula de la misma forma en cada uno de los estados futuros como puede verse en la ecuación extendida 2.9 y así recursivamente. Esta recursividad hace imposible el cálculo de la ecuación.

La programación dinámica trata de resolver este problema mediante algoritmos iterativos como el algoritmo de *iteración de la política*. Este algoritmo consta de dos partes: la evaluación de la política y la mejora de la política. Durante la evaluación de la política, el algoritmo aplica la función de valor $V(s)$ recursiva de acuerdo a la política actual pero acaba cuando el incremento Δ de valor de la iteración actual respecto a la anterior, $\Delta = V^{\pi_{k-1}}(s) - V^{\pi_k}(s)$, no supera cierto parámetro θ[3], $\Delta < \theta$. Luego, el algoritmo trata de mejorar la política basándose en la función de valor anterior. Si la política no difiere estamos ante una política estable (converge) y finaliza la repetición.

En la práctica, el algoritmo de iteración de la política converge en un número reducido de iteraciones, aunque el tiempo de convergencia es prohibitivamente grande cuando el espacio de estados es grande. Además, este algoritmo requiere un conocimiento de las función de transición T y la función de refuerzo R (García, 2012).

[3]Normalmente un número positivo pequeño.

Existen otros algoritmos iterativos como el algoritmo de *iteración del valor*, en el que, en lugar de evaluar la política actual para encontrar una política mejor a partir de ese valor, se trata de encontrar la función de valor óptima, en un proceso con la misma condición de finalización a partir del parámetro θ, para, a partir de es función de valor óptima, extraer una política (que también debería ser óptima, es decir, convergente).

En cualquier caso, estos algoritmos de programación dinámica tratan de obtener una política de comportamiento óptima a partir de funciones de valor calculadas en un subconjunto del problema, en lugar del problema en todo su conjunto (sería intratable). Además, se necesita conocer todo el modelo del entorno para ser aplicables.

Pero no siempre podemos conocer todo el modelo. De hecho, en la mayoría de los casos el modelo de transición de estados y la función de recompensa serán elementos desconocidos y el agente inteligente tendrá que hacer pruebas con el entorno.

2.3.3. Desconocimiento del modelo del entorno

En ambientes donde se desconoce parcialmente o por completo el modelo a tratar, el agente tiene que interactuar de forma activa con el ambiente para extraer esa información que no conoce. Existen dos enfoques: por un lado, el agente puede tratar de aproximarse al modelo mediante estadística para así poder aplicar las técnicas de programación dinámica explicadas en el apartado anterior 2.3.2, o, por el contrario, puede prescindir del modelo y aplicar técnicas que permiten al agente solucionar el problema a partir de los ejemplos de las distintas ejecuciones realizadas sobre el entorno.

Aquí entran dos conceptos importantes del aprendizaje por refuerzo: la exploración y la explotación. En la mayoría de los casos el sistema inteligente usará la **explotación**, esto es, seguirá la política π actual que indica la acción que hasta el momento se ha aprendido que es la mejor. Pero en ciertas ocasiones el sistema inteligente usará **exploración**, en otras palabras, realizará una acción aleatoria para ver si, a la larga, le puede llevar a obtener una solución alternativa que mejore las recompensas.

En RL existen muchas técnicas que combinan la explotación y la exploración para resolver problemas en los que se desconoce la dinámica del entorno. Un ejemplo de estas técnicas son los *métodos de Montecarlo*. Estos métodos requieren de ejemplos de secuencias de acciones, estados y recompensas en interacciones reales con el ambiente. Se interacciona desde un estado inicial y a partir de ahí se realizan acciones que llevan a nuevos estados hasta que eventualmente e independientemente de las acciones realizadas se llega a un estado terminal que reinicia al agente de nuevo al estado inicial. A esta sucesión de pares estado-acción que ocurren desde un estado inicial a un estado final se le llama *episodio*. Los métodos de Montecarlo solamente modifican

las políticas y las estimaciones de valor después de completar un episodio.

Para estimar la función de valor de estado para cierta política, a través de la experiencia, simplemente se calcula el promedio de los refuerzos. Por ejemplo, calcular el valor de un estado concreto $V^\pi(s)$ trataría de promediar las recompensas obtenidas cada vez que ha aparecido el estado s en un episodio. A cada ocurrencia del estado s en un episodio se le llama una *visita* de s. Por supuesto, s puede ser visitado varias veces en el mismo episodio. El método de Montecarlo de la primera visita, del inglés *first-visit MC method*, estima $V^\pi(s)$ como el promedio de las recompensas después de las primeras visitas a s, mientas que el método de Montecarlo de cada visita, del inglés *each-visit MC method*, promedia todas las recompensas obtenidas en todas las visitas a s. Estos dos métodos de Montecarlo son muy similares pero tienen propiedades teóricas ligeramente diferentes (Sutton y Barto, 1998).

Tanto el método de la primera visita como el de cada visita convergen a $V^\pi(s)$ ya que el número de visitas (o primeras visitas) tiende a infinito. En el caso del método de la primera visita, la desviación estándar del error en la estimación de las recompensas cae como $1/\sqrt{n}$, donde n es el número de recompensas promediadas. El método de cada visita es menos directo, pero sus recompensas también convergen cuadráticamente a $V^\pi(s)$ (Singh y Sutton, 1996).

En los problemas en los que no disponemos del modelo, es más útil estimar la función de valor de acción Q^π. Con un modelo, es suficiente aplicar programación dinámica para determinar una política mediante la función de valor $V(s)$ como se explica en el apartado 2.3.2, pero sin un modelo los valores de estado por sí solos no son suficientes. Es necesario estimar explícitamente el valor de cada acción para que los valores sean útiles al sugerir una política. Al igual que antes, se utilizan los métodos de la primera visita y de cada visita, pero ahora para estimar el valor de cada par estado-acción en lugar de un estado. Se dice que un par estado-acción ha sido visitado si al encontrarte en un estado se ha realizado la acción. El método de la primera visita promedia las recompensas obtenidas la primera vez en cada episodio en que se visitó el estado y se realizó la acción. El método de cada visita promedia las recompensas obtenidas en todas las visitas de cada par estado-acción. Estos métodos convergen de forma cuadrática, como antes, a medida que el número de visitas a cada par estado-acción se aproxima al infinito.

El problema a la hora de estimar Q^π es que puede darse que pares estado-acción nunca sean visitados. Obviamente, esto es un problema, ya que el objetivo de calcular los valores de acción es crear una política que nos ayude a elegir la mejor acción para cada estado. La manera de garantizar que todos los pares estado-acción sean visitados es la exploración continua. Lo que se suele hacer es considerar solamente aquellas políticas estocásticas[4]

[4]El término estocástico hace referencia al estado de un sistema que actúa con cierto grado de aleatoriedad, por lo que lo convierte en no determinista.

que tienen una probabilidad distinta a cero de seleccionar todas las acciones en cada estado.

Como en programación dinámica, Montecarlo también tiene un proceso de mejora de políticas. Tras cada episodio las recompensas observadas se usan para evaluar la política y la política se mejora para todos los estados visitados en el episodio. Tenemos, entonces, que para cada estado s en el episodio la política se mejora como se muestra en la ecuación

$$\pi(s) = \arg\max_a Q(s,a) \tag{2.10}$$

2.3.4. Métodos de diferencias temporales

Los métodos más utilizados en RL para problemas en los que existe un desconocimiento de la dinámica del entorno son los métodos de diferencia temporal (del inglés *temporal difference methods*), a partir de ahora TD. Estos métodos son una combinación de la programación dinámica y de los métodos de Montecarlo. Por un lado, los métodos de TD actualizan estimaciones basadas en parte de otras estimaciones aprendidas, sin esperar al resultado. Por otro lado, TD aprende de la experiencia adquirida de interactuar directamente con el entorno como hacen los métodos de Montecarlo.

Dentro de TD, el algoritmo más desarrollado tanto en la teoría como en la práctica es *Q-Learning*. Este algoritmo calcula la política óptima sólo a partir de la experimentación. La principal diferencia con Montecarlo es que no tiene que esperar al final de un episodio para actualizar los valores sino que lo hace al finalizar cada paso. Como se ha explicado en el apartado anterior 2.3.3, en entornos en los que se desconoce el modelo es más útil estimar la función de valor de acción y por eso hay una variante de los métodos de Montecarlo que sólo estima Q^π. En este caso, se trata de calcular lo que en Q-Learning se denomina los $q-valores$, que no son otra cosa que los valores de la función Q.

Estos q-valores se calculan a partir de q-valores anteriores y las recompensas obtenidas. Para calcular el q-valor siguiente Q_{k+1} a partir del q-valor que tenemos actualmente Q_k se usa la ecuación 2.9, por lo que tendríamos que

$$Q_{k+1}(s,a) = \sum_{s'} Pr^a_{s,s'}[R(s,a) + \gamma \max_{a' \epsilon A(s)} Q_k(s',a')] \tag{2.11}$$

El problema aquí es que, al no conocer el modelo del entorno, no conocemos las probabilidades $Pr^a_{s,s'}$ ni la función de recompensa $R(s,a)$, por lo que necesitamos sustituirlas en la ecuación. En el caso de la función de recompensa es simple: el agente ejecuta la acción sobre el estado y se sustituye por la recompensa r concreta que devuelva el ambiente al ejecutar esa acción, tal que

$$Q_{k+1}(s,a) = \sum_{s'} Pr^a_{s,s'}[r + \gamma \max_{a' \epsilon A(s)} Q_k(s',a')] \tag{2.12}$$

Para poder sustituir la función de transición de estados lo que se hace es un muestreo, esto es, una combinación ponderada del q-valor actual y el nuevo q-valor obtenido. Si en la ecuación 2.12 la muestra δ es

$$\delta = r + \gamma \max_{a' \epsilon A(s)} Q_k(s', a') \tag{2.13}$$

La combinación ponderada vendría definida como

$$Q_{k+1}(s, a) = (1 - \alpha) Q_k(s, a) + \alpha\delta \tag{2.14}$$

Donde α es una tasa de aprendizaje, comprendida entre 0 y 1, que se encarga de decidir cuánto ponderamos el q-valor que ya conocemos respecto al que acabamos de aprender. Si, por ejemplo, $\alpha = 0'9$ entonces lo aprendido se pondera 0'9 y lo ya conocido $1 - \alpha = 0'1$.

La ecuación final es de la forma

$$Q_{k+1}(s, a) = (1 - \alpha) Q_k(s, a) + \alpha[r + \gamma \max_{a' \epsilon A(s)} Q_k(s', a')] \tag{2.15}$$

Es importante calcular todos los $Q(s, a)$ por lo que hay que mantener una exploración sobre todo en los momentos iniciales del entrenamiento. Sin exploración no se puede llegar a la política óptima porque estaríamos dejando sin probar acciones que podrían ser las mejores. Para decidir cuándo explorar nuevas acciones o cuándo explotar las ya conocidas se puede utilizar la técnica ε-*greedy* en la que se realiza una exploración de forma aleatoria según una pequeña probabilidad ε. El problema es que una vez explorado todo y se haya aprendido la política óptima, se sigan realizando exploraciones. Es conveniente, pues, que ε vaya decreciendo con el tiempo, de manera que se realice mucha exploración al principio y, durante el entrenamiento, cada vez se realice menos exploración y más explotación hasta que, cuando se haya llegado a la política óptima, el agente solamente realice explotación.

Capítulo 3

Objetivos

El propósito de este trabajo es investigar y estudiar técnicas de aprendizaje automático que puedan ser aplicables para el desarrollo de un agente inteligente que sea capaz de resolver videojuegos arcade emulados.

A continuación se explican más detalladamente los objetivos a cumplir para la consecución del trabajo.

- **Realizar el estudio de plataformas de desarrollo de agentes para videojuegos.** Se utilizará una de las plataformas estudiadas para que nos proporcione un entorno con juegos ya emulados y así poder centrar el trabajo únicamente en el desarrollo del agente. El estudio de estas plataformas viene explicado en la sección 2.2.
- **Analizar técnicas de Inteligencia Artificial aplicables en este contexto.** Estas técnicas se enmarcan dentro del área del aprendizaje reforzado. El análisis de estas técnicas se encuentra en la sección 2.3.
- **Diseñar e implementar el agente inteligente,** aplicando las técnicas analizadas. El agente a implementar debe ser de propósito general, es decir, debe ser aplicable a cualquier juego simulado sobre la plataforma de desarrollo escogida. Los detalles del desarrollo del agente se pueden ver en el capítulo 5. Se realizarán experimentos del agente en varios juegos y se analizarán los resultados, que se encuentran en la sección 5.4.

Capítulo 4

Metodología

En este capítulo se explica la metodología seguida en la realización del presente trabajo. Se han identificado una serie de fases que se han considerado necesarias para cumplir con los objetivos propuestos. Las fases se han completado de forma iterativa, teniendo en cuenta que para poder afrontar una fase se necesitaba haber completado la anterior. También se describen en este capítulo las herramientas que han sido utilizadas para el desarrollo del agente inteligente.

4.1. Fases del trabajo

Durante la ejecución de cada una de las fases se ha seguido la correspondiente tutorización por parte de las coordinadoras del trabajo, realizándose reuniones para revisar el progreso de la fase. En cada una de estas reuniones, para cada fase, se han detectado cambios necesarios, mejoras a realizar y se ha verificado si la fase ha sido completada para poder así hacer efectiva la siguiente fase.

A continuación se enumeran las fases identificadas que se han llevado a cabo durante el trabajo:

1. Estudio previo del marco teórico relacionado con el problema planteado y afianzamiento de los conocimientos necesarios para la elaboración del trabajo.

2. Búsqueda de plataformas de desarrollo en las que poder desarrollar el agente inteligente, así como la investigación de trabajos similares que ayudasen a escoger la plataforma y las técnicas a utilizar.

3. Desarrollos previos en subproblemas más sencillos al problema planteado para consolidar los conocimientos de la técnica utilizada.

4. Diseño e implementación del agente final.

5. Experimentación y análisis de resultados del agente sobre varios entornos de emulación de juegos arcade.

6. Cierre y conclusiones del trabajo.

4.2. Herramientas utilizadas

A continuación se describen las diferentes herramientas y tecnologías que se han utilizado para llevar a cabo el diseño e implementación del agente inteligente:

Plataforma De las plataformas estudiadas se ha hecho uso de OpenAI Gym. Se ha escogido esta plataforma porque cuenta con una colección extensa y diversa de entornos que permiten probar el agente en ambientes menos complejos que un juego Atari. Esto ha permitido realizar la fase 3 del trabajo.

Lenguaje de programación Se ha utilizado Python para implementar el agente inteligente. Es el lenguaje más utilizado en el mundo de la IA, especialmente en machine learning. Su popularidad en este campo se debe a la facilidad que ofrece para realizar cualquier tarea, debido al gran número de bibliotecas con las que cuenta. Realizar algo en Python requiere menos tiempo y líneas de código que en otros lenguajes, algo fundamental en aprendizaje automático donde es más importante el entrenamiento y la experimentación que la implementación en sí.

Librerías De entre todas las bibliotecas o librerías de las que se dispone en Python, se han empleado las siguientes: `Numpy`, una biblioteca de funciones matemáticas que facilitan el tratamiento de vectores, matrices y arreglos multidimensionales; y `Matplotlib`, una potente librería para la creación e impresión de gráficas a partir de datos a fin de presentar los resultados.

Framework de desarrollo Para la implementación del agente inteligente se ha empleado Tensorflow, un completo entorno de trabajo para desarrollar y entrenar modelos de machine learning. Permite paralelizar el flujo de operaciones, depurar durante el entrenamiento y desplegar en la nube. Además, se ha empleado también Keras, que facilita la construcción de modelos en Tensorflow gracias a sus intuitivas APIs de alto nivel. La biblioteca Keras fue diseñada en un principio como una capa de abstracción que utilizaba Tensorflow como *backend*, para que la construcción de redes neuronales fuese todavía mas amigable, modular y extensible. En Tensorflow 2.0 se ha integrado Keras como parte del *core*, por lo que ambas bibliotecas pueden ser utilizadas de forma integrada sin necesidad de ser instaladas por separado.

Capítulo 5

Desarrollo del agente

En este capítulo se explica con detalle el desarrollo seguido para el diseño e implementación del agente inteligente. Es importante remarcar que el proceso de desarrollo del agente ha sido iterativo y evolutivo, de manera que se ha empezado por la implementación de un algoritmo base capaz de resolver problemas sencillos y se ha ido modificando y adaptando a medida que el agente se ha enfrentado a problemas más complejos.

En las primeras secciones del capítulo se explica el algoritmo de partida, que ha sido probado sobre entornos más sencillos al de los videojuegos. Este algoritmo ha ido evolucionando hasta el agente final, explicado en las secciones finales, con el que se ha experimentado en algunos juegos Atari.

5.1. Q-Learning

El algoritmo de partida utilizado para la creación del agente inteligente es el método de aprendizaje por refuerzo Q-Learning. Este método consiste en utilizar la experiencia adquirida a partir de la exploración del entorno para aproximar la política óptima π^*. Para cada paso ejecutado por el algoritmo la política es mejorada tomando como base los valores de la función Q, los q-valores. Partiendo de la ecuación 2.10, si los valores aprendidos durante el entrenamiento se aproximan a los de la función de valor óptima Q^* entonces podemos garantizar que la política que se obtiene es también óptima y se denotaría como

$$\pi^*(n) = \arg\max_a Q^*(s, a) \tag{5.1}$$

5.1.1. Entrenamiento fuera de política

Una característica importante del algoritmo Q-Learning es que la política que se está evaluando y mejorando no es la misma política que se sigue para realizar acciones sobre el ambiente, por lo que se dice que Q-Learning es un método fuera de política o en inglés *off-policy*.

Cuando se realiza una acción sobre un estado, el algoritmo actualiza el q-valor de ese estado a partir del q-valor del estado siguiente y la mejor acción que podría haber realizado según la política, aunque no sea la acción que realmente haya realizado. El agente puede haber realizado una acción aleatoria pero la política siempre se actualiza teniendo en cuenta las acciones que maximicen los q-valores. En resumen, la acción tomada no se tiene en cuenta para mejorar la política ni viceversa. Si el agente siempre siguiese la política para realizar una acción quedaría descartada la exploración, parte importante del algoritmo.

Concretamente, la política que se va a seguir en un principio para realizar acciones es la de tomar una acción aleatoriamente con probabilidad ε y, en caso contrario, tomar la mejor acción aprendida hasta el momento.

5.1.2. Tabla Q

La manera en la que el algoritmo Q-Learning trabaja con cada par estado-acción se puede imaginar como una tabla, a la que llamaremos tabla Q, en la que las filas representarían todos los estados posibles y las columnas cada posible acción que se puede llevar a cabo sobre los estados. Así tenemos que el elemento (i, j) de la tabla se corresponde con el valor de, estando sobre el estado s_i, tomar la acción a_j.

Esta tabla se inicializa arbitrariamente al principio, por ejemplo con todos los valores a 0, y se va actualizando con el entrenamiento.

Tabla Q	a_0	a_1	a_2	a_3	a_4
s_0	$Q(s_0, a_0)$	$Q(s_0, a_1)$	$Q(s_0, a_2)$	$Q(s_0, a_3)$	$Q(s_0, a_4)$
s_1	$Q(s_1, a_0)$	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$Q(s_1, a_3)$	$Q(s_1, a_4)$
s_2	$Q(s_2, a_0)$	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$Q(s_2, a_3)$	$Q(s_2, a_4)$

Tabla 5.1: Estructura de una tabla Q.

Cada vez que el agente realiza la acción a sobre el estado s el q-valor correspondiente $Q(s, a)$ se actualiza en la tabla aplicando la ecuación 2.15.

5.1.3. Decreciendo la exploración

Como ya se ha mencionado anteriormente, una parte fundamental del algoritmo Q-Learning es realizar exploración. Esto es así porque si se usase siempre explotación de lo ya aprendido el agente dejaría de ejecutar algunas acciones sobre algunos estados. La política obtenida dependería entonces enormemente de las primeras ejecuciones realizadas. Por tanto, sin exploración no podemos llegar a la política óptima.

La técnica seguida para que el agente mantenga la exploración en el entrenamiento es *ε-greedy*. Esta técnica consiste en decidir la probabilidad

de realizar exploración o explotación a partir de un pequeño parámetro ε. Se lanza un número *random* entre 0 y 1, si se cumple que el número obtenido es menor que ε entonces se realiza exploración y si es mayor entonces se realiza explotación.

A medida que avanza el entrenamiento se van reduciendo las acciones sin explorar, por lo que necesitamos que la probabilidad de exploración sea menor con el paso del tiempo. Por eso el valor de ε no es constante sino que varía tras cada episodio. En un principio, la probabilidad de explorar se requiere que sea mayor y el valor de ε estará más cercano a 1. Para ir reduciendo esta probabilidad, tras cada episodio el valor de ε se decrece respecto al número de episodios que quedan por ejecutar. De esta manera, ε va decreciendo prograsivamente hasta 0.

Q-Learning (S,A,γ,α,ε) $\rightarrow \pi \approx \pi^*$	
01	**Parámetros:**
02	El conjunto de estados S.
03	El conjunto de acciones A.
04	Factor de descuento $\gamma \epsilon (0, 1]$.
05	Tasa de aprendizaje $\alpha \epsilon (0, 1]$.
06	Número pequeño $\varepsilon > 0$.
07	**Inicializar:**
08	La tabla Q arbitrariamente $\forall s \epsilon S$ y $\forall a \epsilon A : Q(s, a)$
09	**Repetir:**
10	Para todos los episodios
11	Inicializar s
12	Para cada iteración en el episodio
13	Elegir a siguiendo Q con probabilidad ε de exploración
14	Ejecutar a y observar la recompensa r y el nuevo estado s'
15	Actualizar la tabla Q siguiendo
16	$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \, \text{máx}_{a'} \, Q(s', a')]$
17	Pasar al siguiente estado $s \leftarrow s'$
18	hasta que s sea terminal
19	**Devolver:**
20	La política π derivada de Q

Tabla 5.2: Algoritmo Q-Learning.

5.2. Desarrollos previos

La plataforma de desarrollo utilizada para la implementación y experimentación del agente inteligente es OpenAI Gym. Esta plataforma proporciona una gran cantidad de ambientes, no solamente de juegos arcade, sino de diferentes problemas en los que es posible aplicar RL.

Los ambientes en Gym están divididos en categorías, de menor a mayor dificultad, siendo los más fáciles los problemas de la categoría denominada *toy text* que son entornos de texto simples con los que empezar. La siguiente categoría es *classic control*, un conjunto de problemas estándar en la literatura del aprendizaje reforzado.

Se han escogido dos problemas de cada una de estas categorías (*FrozenLake*, de la categoría *toy text*, y *MountainCar*, de la categoría *classic control*) con los que empezar a probar el algoritmo antes de enfrentarlo a juegos arcade.

5.2.1. FrozenLake

El primer entorno utilizado para empezar a desarrollar el agente inteligente es *FrozenLake*, un sencillo juego de texto por consola que representa un lago congelado por el cual el agente tiene que desplazarse desde un punto de partida hasta la meta evitando unos agujeros en el hielo por los que se cae al agua helada. La interfaz del juego puede verse en la figura 5.1.

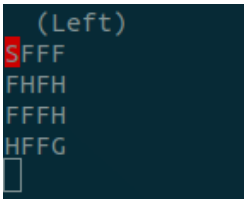


Figura 5.1: FrozenLake en Gym.

En este juego de texto el punto de partida está representado como S y la meta que tiene que alcanzar el agente está representada como G. Los estados seguros por los que puede moverse el agente sin caer al agua se representan como F. Los estados H representan los agujeros en el hielo, si el agente cae en uno de estos estados vuelve automáticamente al punto de partida finalizando así el episodio.

Este juego es fácil de resolver porque posee un conjunto muy reducido de estados, concretamente dieciséis posibles estados en los que puede encontrarse el agente. A su vez, solamente hay cuatro posibles acciones: mover arriba, abajo, a la izquierda o a la derecha. Entonces, la tabla Q necesaria para resolver este problema es de tamaño 4x16. Las recompensas recibidas por el agente en este juego son de 1 cuando llega a la meta y 0 en caso contrario.

Este juego se resuelve simplemente aplicando el algoritmo Q-Learning tal y como está descrito en la tabla 5.2.

5.2.2. MountainCar

Un problema clásico y muy estudiado en RL es *MountainCar*, que consiste en un coche situado entre dos montañas que tiene como objetivo alcanzar la cima de la montaña de la derecha (ver figura 5.2). El motor del coche no es lo suficientemente potente para subir la montaña por lo que se necesita conducir hacia delante y hacia atrás para acumular impulso.

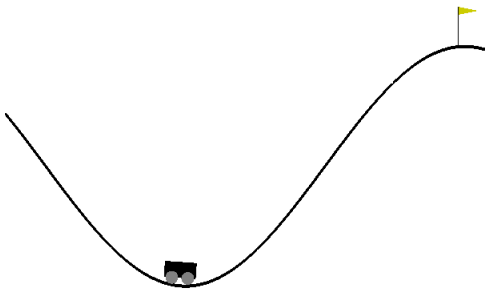


Figura 5.2: MountainCar en Gym.

El coche tiene 200 movimientos como máximo para conseguir el objetivo, de lo contrario el episodio finaliza. Hay tres acciones disponibles: avanzar, retroceder o no hacer nada. En este caso, las recompensas son siempre negativas, obteniéndose -1 puntos por cada movimiento realizado así que el objetivo tiene que ser alcanzado en el menor número de movimientos. En el peor de los casos, se realizarán 200 movimientos sin alcanzar el objetivo y la recompensa total acumulada será de -200 y, en el mejor de los casos, se llegará al objetivo en un movimiento y el total de recompensas será -1.

La diferencia entre este juego y el anterior está en el número de estados posibles en el que puede encontrarse el agente. Aquí el conjunto de estados es representado como un vector de dos dimensiones, donde la primera dimensión representa la posición en la que se encuentra el coche en la curva que dibujan las montañas y la segunda dimensión representa la velocidad a la que se mueve el coche. La posición puede ser cualquier valor en el rango de -1'2 a 0'6 y la velocidad puede ser cualquier valor en el rango de -0'07 a 0'07. Esto convierte al conjunto de estados en un espacio de estados continuos e imposibilita la creación de una tabla Q para todos los estados.

Para resolver este problema es necesario **discretizar el espacio de estados**. Para ello, se redondea cualquier valor de posición a su décima más cercana y cualquier valor de velocidad a su centésima más cercana. Así infinitos valores son agrupados en un mismo número. Esta división en grupos o regiones reduce el número de estados considerablemente, ya que todos los estados que se encuentran dentro de cada una de esas regiones son tratados como si se tratasen del mismo estado. Por tanto, el conocimiento que se obtenga para cualquier estado de cualquier región, puede generalizarse a

cualquier estado de su misma región (Fernández, 2002). Este juego puede resolverse aplicando este método, sin embargo, para ambientes con un conjunto de estados mucho más elevado o con más de dos dimensiones no llega a ser suficiente.

5.3. DQN

En la siguiente categoría enfrentamos al agente inteligente a juegos arcade, que es el objetivo del presente trabajo. En Gym, todos los entornos de esta categoría son videojuegos Atari. En estos ambientes, cada observación es una imagen RGB de la pantalla y es representado como una matriz de tres dimensiones de tamaño 210x160x3.

Como se ha explicado anteriormente, para un espacio de estados continuo no es posible la creación de una tabla Q porque existen infinitos estados posibles. Hay que reducir el número de estados, por ejemplo discretizando el espacio de estados, para poder aplicar la tabla Q. Pero en este caso el número de estados es tan elevado que para discretizar el espacio en un conjunto de estados tratable se tendría que considerar por igual demasiados estados diferentes. Es por ello necesario sustituir la tabla Q en el algoritmo Q-Learning. Existe una evolución del algoritmo llamada Deep Q-Network (DQN) que encuentra una alternativa a la tabla Q para calcular los q-valores.

El algoritmo DQN utiliza redes neuronales convolucionales para tratar cada imagen del juego, que representa el estado actual. En este caso, en lugar de hacer uso de la tabla Q para actualizar el q-valor de la acción realizada sobre un estado, la red neuronal calcula un q-valor para cada acción posible sobre ese estado (ver figura 5.3).

Por tanto, en DQN tenemos una red neuronal cuyos valores de entrada son los estados del juego Atari que vienen representados como un fotograma del juego y tendrá tantas neuronas de salida como acciones se pueden tomar en el juego.

5.3.1. Construyendo el estado

Como ya se ha indicado, el estado de un juego Atari es la imagen del juego. El problema es que una imagen por sí sola no muestra toda la información del estado. Un estado es toda la información relativa a la situación en la que se encuentra el agente respecto al entorno. Hay ciertos aspectos de esta información que es imposible obtener de una imagen estática. Por ejemplo, supongamos que en nuestro juego tenemos que golpear una pelota que está en movimiento. Observando una imagen estática, ¿cómo podemos saber si la pelota se está moviendo hacia la izquierda o hacia la derecha? Del mismo modo, ¿cómo podemos averiguar a qué velocidad se está moviendo? Para solucionar este problema, un estado será representado como una

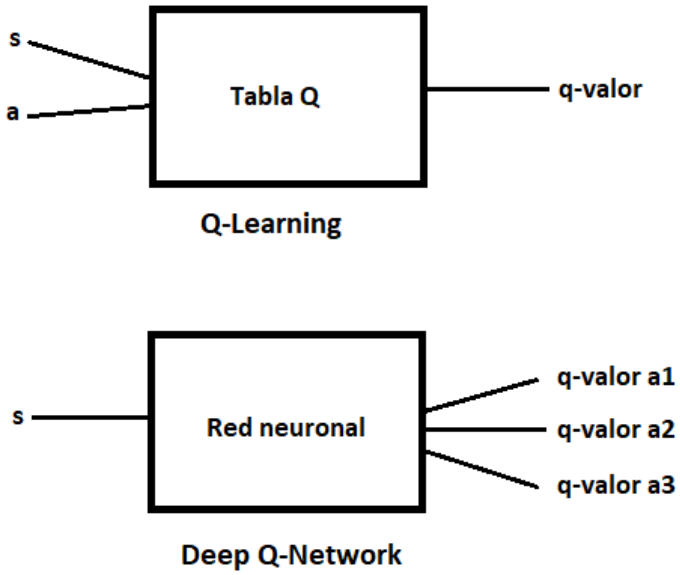


Figura 5.3: Comparativa Q-Learning y DQN.

secuencia de x imágenes consecutivas y la acción a ejecutada sobre cada imagen, $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$. Concretamente, se ha creado una secuencia por cada cuatro fotogramas observados.

Además, a toda esta secuencia se le aplica un preprocesado ϕ para reducir la dimensionalidad de la entrada a la red neuronal, ya que el tamaño original de las imágenes puede ser exigente en términos de cómputo y requisitos de memoria. La matriz de valores de cada imagen se ha reducido a tamaño 84x84x4.

5.3.2. Arquitectura de la red neuronal

A continuación, se explica brevemente la arquitectura de la red neuronal que se ha construido, la cual fue diseñada por Mnih et al. (2015), creadores del algoritmo DQN, y puede verse representada en la figura 5.4.

La capa de entrada de la red neuronal recibe la secuencia de entrada de tamaño 84x84x4 construida por la función de preprocesamiento ϕ explicada anteriormente. Esta entrada es tratada por la primera capa oculta de la red, una capa convolucional, que consiste en 32 filtros de 8x8 con *stride* 4 y aplica la función de activación ReLU.

La segunda capa oculta, también convolucional, involucra 64 filtros de 4x4 con *stride* 2 y aplica nuevamente la función de activación ReLU. Esto es seguido por una tercera capa convolucional que involucra 64 filtros de 3x3 con *stride* 1 seguido de la misma función de activación que el resto de capas.

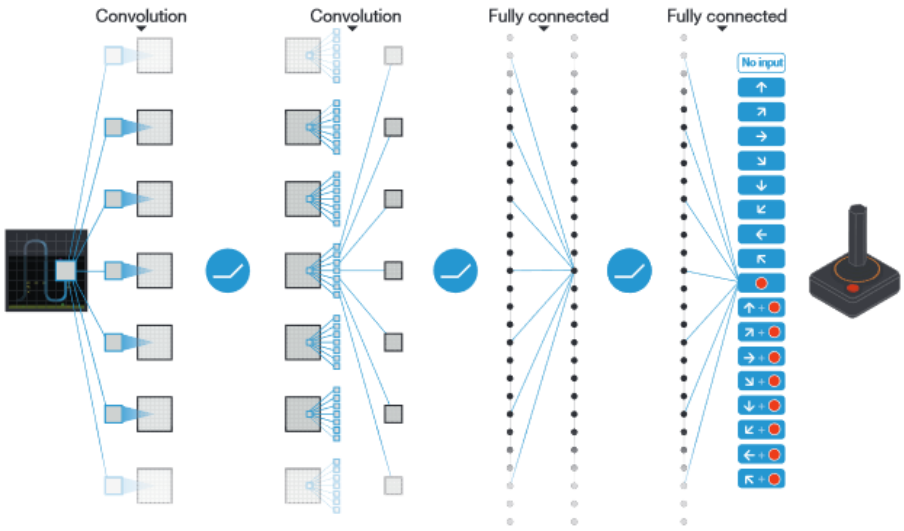


Figura 5.4: Ilustración de la arquitectura de la red.

La capa oculta final es una capa totalmente conectada formada por 512 neuronas rectificadoras. Esta capa está conectada con la capa de salida, consistente en una capa clasificadora totalmente conectada que tiene tantas neuronas como acciones posibles en el juego. El valor de cada neurona de salida es el q-valor de la acción correspondiente.

Esta clasificación sustituye entonces la ecuación 2.15 que utiliza Q-Learning para aproximar Q^*. En este caso, en cada iteración i hay que minimizar la función de coste $L_i(\theta_i)$ obtenida durante el entrenamiento de la red, de la siguiente manera:

$$L_i(\theta_i) = E\left[(y_i - Q(s, a; \theta_i))^2\right] \tag{5.2}$$

Donde θ_i representa los pesos de la red neuronal en la iteración i. El valor y_i es el correspondiente a cada neurona de salida, es decir, el q-valor, que se obtiene aplicando $y_i = E[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]$. Los parámetros de la iteración anterior θ_{i-1} se mantienen fijos al optimizar la función de coste $L_i(\theta_i)$. Los valores de salida y_i dependen de los pesos de la red, a diferencia de lo que ocurre en aprendizaje supervisado, donde los valores de salida se fijan antes de empezar el entrenamiento y los pesos se ajustan para llegar a esos valores. Diferenciando la función de pérdida con respecto a los pesos llegamos al siguiente gradiente,

$$\nabla_{\theta_i} L_i(\theta_i) = E\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{5.3}$$

Se ha demostrado que el uso de redes neuronales en RL puede provocar inestabilidad y divergencia en los resultados (Mnih et al., 2013). Esto es debido a las correlaciones presentes en las secuencias de observaciones. Por lo tanto, es importante encontrar un método que evite el aprendizaje mediante muestras consecutivas que ha demostrado ser ineficiente.

5.3.3. Experience replay

Existe una técnica llamada *experience replay* que consigue aleatorizar de forma eficiente las muestras con las que entrena la red neuronal, lo que nos permite entrenarla sin divergencia. Esta técnica consiste en crear un conjunto de experiencias e en cada paso de tiempo t, de manera que cada experiencia almacena una transición de una secuencia, ejecutar una acción, recibir una recompensa y obtener una nueva secuencia; $e_t = (s_t, a_t, r_t, s_{t+1})$. Estas experiencias se almacenan en un conjunto de datos D denominado memoria *replay*, $D_t = \{e_1, ..., e_t\}$, durante un número de episodios N.

Durante el entrenamiento, se realizan actualizaciones en Q sobre muestras de experiencia, $(s, a, r, s') \sim U(D)$, extraídas al azar del conjunto de muestras almacenada. Mediante el uso de *experience replay* la distribución del algoritmo se promedia en muchos de sus estados anteriores, suavizando el aprendizaje y evitando oscilaciones o divergencias en los parámetros. La forma final del algoritmo DQN, aplicando *experience replay*, está descrito en la tabla 5.3.

	DQN (experience replay)
01	**Inicializar:**
02	Memoria *replay* D con capacidad N
03	La red Q con pesos θ aleatorios
04	**Repetir:**
05	Para todos los episodios
06	Inicializar una secuencia $s_1 = x_1$ y preprocesarla $\phi_1 = \phi(s_1)$
07	Para cada iteración en el episodio
08	Elegir a siguiendo Q con probabilidad ε de exploración
09	Ejecutar a y observar la recompensa r y la imagen x_{t+1}
10	Crear $s_{t+1} = s_t, a_t, x_{t+1}$ y preprocesar $\phi_{t+1} = \phi(s_{t+1})$
11	Almacenar transición $(\phi_t, a_t, r_t, \phi_{t+1})$ en D
12	Muestra aleatoria de transiciones $(\phi_j, a_j, r_j, \phi_{j+1})$ de D
13	Establecer $y_j = \begin{cases} r_j & \textit{si } \phi_{j+1} \textit{ es terminal} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \textit{en otro caso} \end{cases}$
14	Aplicar descenso por grandiente en $(y_j - Q(\phi_j, a_j; \theta))^2$
15	siguiendo la ecuación 5.3

Tabla 5.3: Algoritmo DQN.

5.3.4. Entrenamiento

El algoritmo DQN está diseñado para ser genérico, esto es, para aprender a jugar a cualquier juego Atari sin necesidad de realizar modificaciones específicas en el modelo para según qué juegos. Por lo tanto, en todos los juegos en los que se ha probado el agente inteligente se ha utilizado la misma arquitectura de red, algoritmo de aprendizaje y configuración de hiperparámetros.

Hay que tener en cuenta también que, al requerirse cuatro fotogramas para construir la secuencia que representa el estado, el agente selecciona una acción cada cuatro fotogramas. Durante los fotogramas omitidos el agente repite la última acción de seleccionó.

Como la escala de recompensas varía mucho de un juego a otro, durante el entrenamiento se ha aplicado una modificación de las recompensas recibidas por el agente. Todas las recompensas positivas se han transformado en 1 y todas las recompensas negativas en -1, dejando las recompensas que son 0 sin modificar. Esto se ha hecho para limitar la escala de los derivados de error y facilitar el uso de la misma tasa de aprendizaje. Como función de optimización se ha utilizado RMSProp.

Para la gestión de la exploración se ha mantenido la estrategia ε-*greedy*. El entrenamiento ha empezado con un valor ε de 1 y ha ido decreciendo hasta llegar hasta 0'01 con el objetivo de aumentar paulatinamente la probabilidad de realizar explotación, es decir, ejecutar el valor máximo de las salidas de la red.

Se han utilizado los siguientes valores de hiperparámetros:

- Muestras de experiencia de tamaño 32.
- Factor de descuento $\gamma = 0'99$.
- Tasa de aprendizaje $\alpha = 0'00025$.
- Valor de momento $\mu = 0'95$[1].
- Probabilidad de exploración ε entre 1 y $0'1$.
- Una observación de 4 fotogramas.
- Gradiente mínimo cuadrado 0'01 para RMSProp.

[1]El momento es un hiperparámetro que puede acelerar el aprendizaje y reducir el riesgo de que el algoritmo se vuelva inestable. Es utilizado por el optimizador RMSProp.

5.4. Experimentos

Para evaluar al agente inteligente, lo entrenaremos para jugar a varios juegos y analizar la evolución de la recompensa acumulada durante el entrenamiento. Para presentar esta evolución, se calcula la recompensa media acumulada cada diez episodios. Al finalizar el entrenamiento se ejecutará el agente en modo *test* para comprobar si el comportamiento del agente mejora significativamente el azar. Se han realizado experimentos sobre dos juegos: *Breakout* y *Space Invaders*.

Todos los resultados presentados a continuación se han obtenido tras la experimentación sobre una CPU Intel i7 4770k de 3.5 Ghz.

5.4.1. Breakout

En Breakout, el agente debe golpear una pelota dirigiendo una pala con el objetivo de romper unos bloques. En este juego se recibe una recompensa cada vez que se rompe un bloque. Los bloques están organizados en filas consecutivas, siendo las filas más profundas solamente accesibles si se han roto bloques en la fila anterior. La recompensa que se recibe al romper un bloque depende de la fila en la que se encuentre el bloque, siendo los bloques de las filas más profundas los que mayores recompensas otorgan. En la figura 5.5 puede verse una instantánea del agente inteligente jugando a Breakout.

Figura 5.5: Agente jugando a Breakout.

El agente inteligente ha sido entrenado a lo largo de aproximadamente unos seis mil episodios, tras los cuales el agente ha alcanzado una puntuación media de entre 60 y 70 puntos cada diez episodios. En la gráfica de la figura 5.6 está representada la evolución de las recompensas obtenidas durante el entrenamiento. Se observa que el agente tardó mil episodios en empezar a obtener una mejora apreciable, teniendo una evolución lenta al principio, hasta alrededor de 20 puntos donde la velocidad del aprendizaje aumenta.

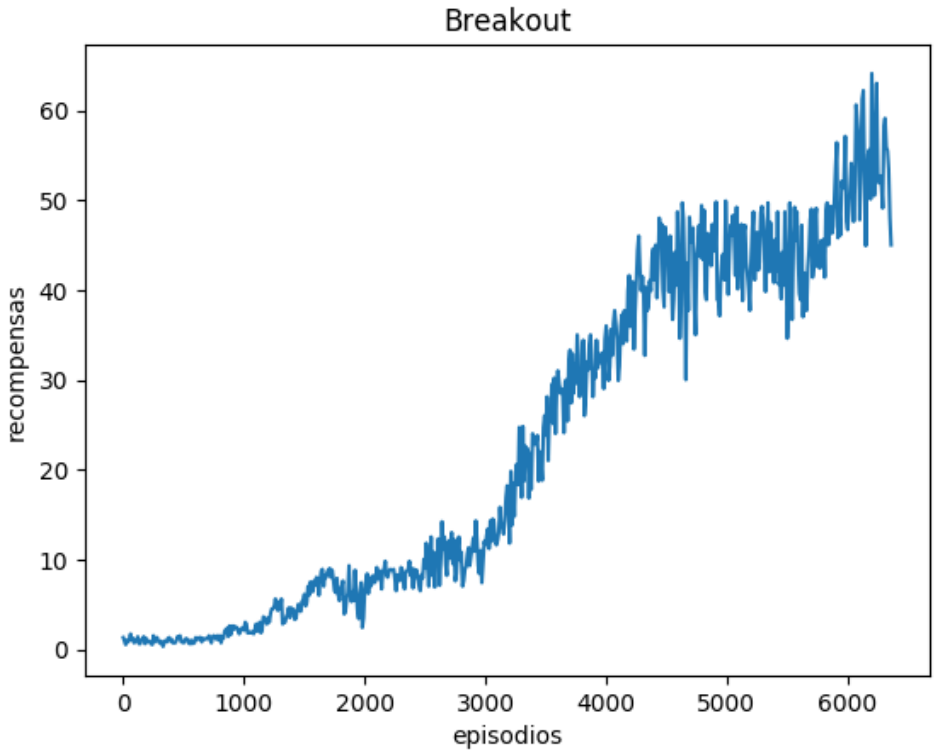


Figura 5.6: Resultado: Evolución de entrenamiento en Breakout.

Para la evaluación del agente inteligente tras el entrenamiento, ha sido ejecutado durante veintidós episodios sobre el modelo entrenado. En todas las ejecuciones de evaluación el agente ha conseguido eliminar la primera fila de bloques al completo y la segunda en la mayoría de los casos. Solamente en una ocasión ha conseguido llegar a golpear la última fila de bloques.

Se ha ejecutado un agente aleatorio el mismo número de episodios con el objetivo de emplearlo como una comparación de referencia. Al comparar el agente entrenado con el agente que realiza acciones aleatorias podemos comprobar cuánta mejora se ha obtenido después del entrenamiento respecto al azar. Esta comparativa se presenta en la gráfica de la figura 5.7. Se puede constatar que en Breakout el agente inteligente ha conseguido mejorar el azar con un margen considerable.

Hay que tener en consideración que, al igual que en el entrenamiento, durante la evaluación el agente también repite la misma acción durante los cuatro fotogramas que se requieren para generar una nueva secuencia de entrada a la red.

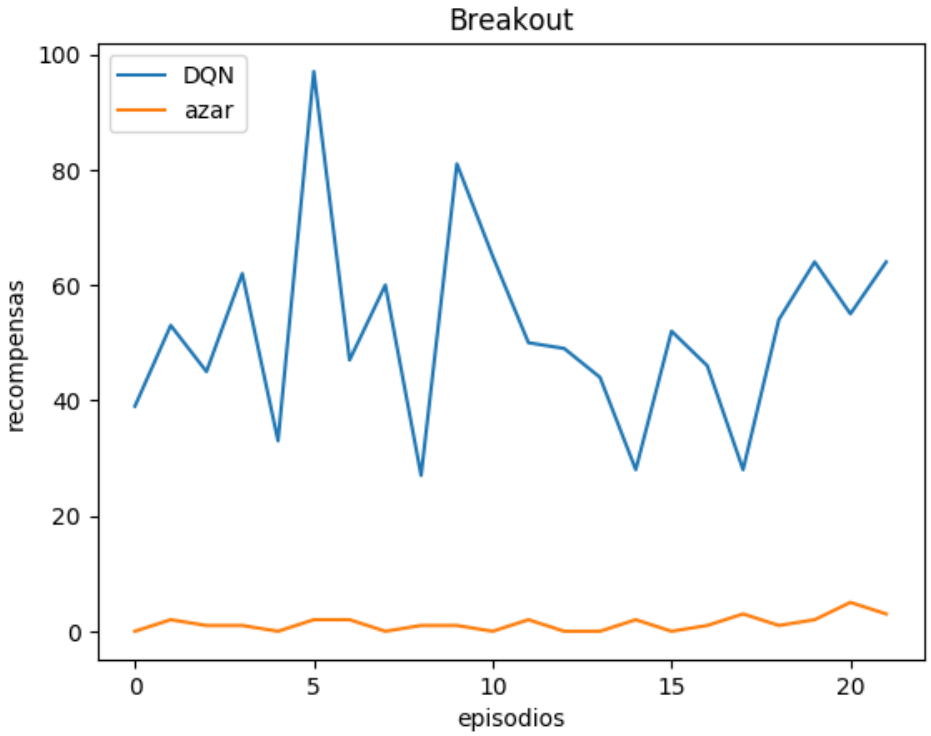


Figura 5.7: Resultado: Mejora de DQN respecto al azar en Breakout.

5.4.2. SpaceInvaders

En Space Invaders, el agente debe eliminar una oleada de alienígenas con un cañón láser antes de ser alcanzado por la propia oleada o por el disparo de uno de los alienígenas. En este juego se recibe una recompensa cada vez que se elimina a un alienígena de la oleada. En la figura 5.8 puede verse una instantánea del agente inteligente jugando a Space Invaders.

Figura 5.8: Agente jugando a Space Invaders.

Tras casi seis mil episodios de entrenamiento, el agente ha alcanzado una puntuación media de alrededor de 500 puntos cada diez episodios. La evolución del entrenamiento está representada en la gráfica de la figura 5.9. A diferencia de Breakout, la evolución de la recompensa media acumulada se mantiene constante durante todo el entrenamiento, sin sufrir aceleraciones. Esta evolución es lenta durante todo el entrenamiento. Esto puede deberse a que la ejecución de acciones aleatorias en este juego puede conllevar la obtención de un alto número de recompensas. El agente puede pensar que está llevando a cabo buenas acciones cuando en realidad no lo son, tardando más en descartarlas.

Existe una diferencia importante entre Breakout y Space Invaders: en Breakout, el agente inteligente tiene que aprender a golpear la pelota con la pala si quiere mantenerse en el juego y poder así acumular recompensas; en Space Invaders, el agente puede mover el cañón láser de izquierda a derecha sin ningún criterio y mantenerse en el juego eliminando alienígenas accidentalmente. Mientras que el objetivo en Breakout es bastante claro: golpear la pelota con la pala; en Space Invaders es algo más difuso: encontrar la manera más eficiente de eliminar alienígenas para que no te alcance la oleada o un disparo.

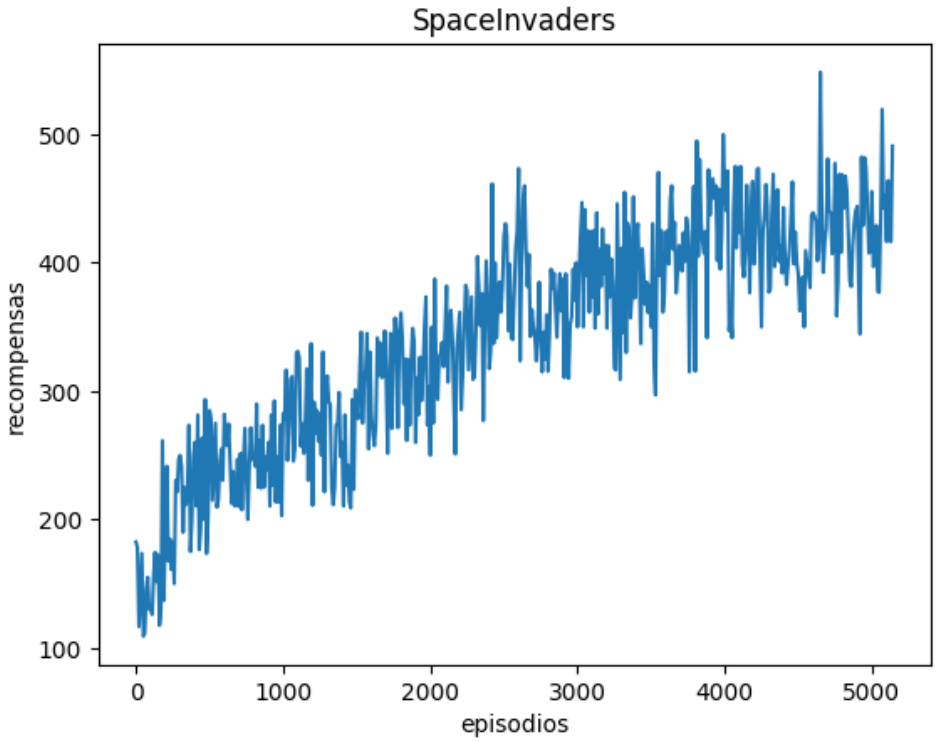


Figura 5.9: Resultado: Evolución de entrenamiento en Space Invaders.

Esta diferencia en el proceso de aprendizaje se ve claramente al comparar el agente entrenado con el agente aleatorio. Esta comparativa se presenta en la gráfica de la figura 5.10. En este caso, el margen de mejora respecto al azar es más discreto.

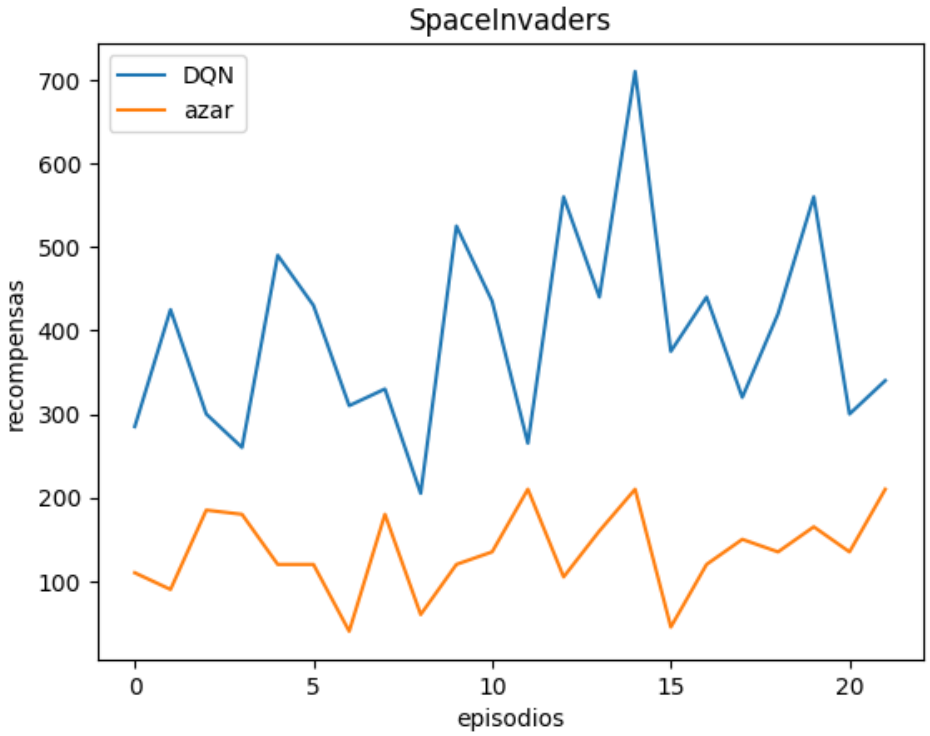


Figura 5.10: Resultado: Mejora de DQN respecto al azar en Space Invaders.

Capítulo 6

Conclusiones

Este trabajo ha puesto en práctica una técnica de aprendizaje reforzado que obtiene buenos resultados en el problema planteado. En trabajos similares en los que se aplica DQN también han conseguido resultados que mejoran el azar e incluso a jugadores humanos expertos, lo que demuestra que el uso de redes neuronales profundas es eficiente en RL siempre y cuando se empleen técnicas que eliminen la divergencia en el entrenamiento. Como es lógico, los resultados presentados en este trabajo se han visto limitados por los recuros de *hardware* utilizados.

La implementación de este algoritmo ha permitido cumplir el objetivo de crear un agente inteligente de ámbito general, empleando el mismo modelo de red en todos los experimentos sin ajustes en la arquitectura o hiperparámetros. No obstante, como puede verse en los resultados de este trabajo, el desempeño que alcanza el agente tras el entrenamiento no es el mismo en todos los juegos. La evolución de la recompensa acumulada depende directamente de la distribución de las recompensas en cada juego. A esta dificultad en el entrenamiento se le llama problema de exploración difícil, del inglés *hard-exploration problem*, y se produce en juegos donde el agente requiere mucha exploración para aprender aquellas acciones que maximicen las recompensas, sobre todo cuando las recompensas son escasas o están muy dispersas. DQN no es eficiente en juegos donde existe este problema.

Como trabajo futuro, sería interesante probar el algoritmo en juegos más complejos que los arcade. La propia OpenAI ha hecho pública una evolución de Gym, la plataforma utilizada en este trabajo, llamada Retro Gym en la que es posible emular juegos de consolas Nintendo como Game Boy o NES y consolas Sega como la Master System.

El código fuente resultante de la realización de este trabajo es accesible desde el siguiente repositorio:

`https://gitlab.com/benjapamies/atari_dqn`

Bibliografía

Bellemare, M. G., Naddaf, Y., Veness, J. y Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, vol. 47, páginas 253–279, 2013.

Bellman, R. Dynamic programming and stochastic control processes. *Information and Control*, vol. 1, páginas 228–239, 1958.

Fernández, F. *Aprendizaje por Refuerzo en Espacios de Estados Continuos*. Tesis Doctoral, Escuela Politécnica Superior. Universidad Carlos III de Madrid, 2002.

García, F. J. *Aprendizaje por Refuerzo para la Toma de Decisiones Segura en Dominios con Espacios de Estados y Acciones Continuos*. Tesis Doctoral, Departamento de Informática. Universidad Carlos III de Madrid, 2012.

Jeerige, A., Bein, D. y Verma, A. Comparison of deep reinforcement learning approaches for intelligent game playing. En *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, páginas 0366–0371. 2019.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J. y Bowling, M. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, vol. 61, páginas 523–562, 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. y Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. y Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, vol. 518(7540), páginas 529–533, 2015. ISSN 00280836.

SINGH, S. P. y SUTTON, R. S. Reinforcement learning with replacing eligibility traces. *Mach. Learn.*, vol. 22(1-3), páginas 123–158, 1996. ISSN 0885-6125.

SUTTON, R. S. y BARTO, A. G. *Reinforcement Learning: An Introduction*. MIT Press, primera edición, 1998.

ZHANG, Y., ZHANG, Q. y YU, R. Markov property of markov chains and its test. En *2010 International Conference on Machine Learning and Cybernetics*, vol. 4, páginas 1864–1867. 2010. ISSN 2160-133X.